Generation of Efficient C Code from MATLAB Using SSA-based Optimizations

Luís Reis
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto, Portugal
ei09030@fe.up.pt

João Bispo
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto, Portugal
jbispo@fe.up.pt

João M. P. Cardoso
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto, Portugal
jmpc@fe.up.pt

Abstract

Many fields of engineering, science and finance use models that are developed and validated in high-level languages such as MATLAB. However, as these systems are moved to environments with resource constraints or portability challenges, these models often have to be rewritten in lower-level languages such as C. Doing so manually is costly and error-prone, but automated approaches tend to generate code that can be substantially less efficient than the handwritten equivalents. Additionally, it is usually difficult to read and improve code generated by these tools.

In this paper, we describe how we improved our MATLAB-to-C compiler, based on the MATISSE framework, to be able to compete with handwritten C code. We describe our new IR and the most important optimizations that we use in order to obtain acceptable performance. We also analyze multiple C code versions to see where our generated code still trails behind and identify a few key improvements that could be made to our optimizers to generate code capable of outperforming handwritten C. We evaluate our results using the *Disparity* benchmark, from the San Diego Vision Benchmark Suite, on a desktop computer and on an embedded device.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors Code generation, Compilers, Optimization, Retargetable compilers

General Terms Algorithms, Performance, Languages

Keywords MATLAB, source-to-source compiler, SSA, optimizing compiler

1. Introduction

MATLAB (MathWorks 2016a) is a high-level programming language that is widely used in several scientific and engineering domains, notably for prototyping programs and algorithms. Often, MATLAB users wish to deploy their programs to devices in which the MATLAB runtime is not available, does not have acceptable performance or consumes too many resources. In these cases, programmers may either spend time and resources to manually rewrite their programs in languages such as C, or rely on automated tools to do so. Unfortunately, generating code as efficient as handwritten C remains a challenge, in part due to the dynamic nature of MATLAB, and in part because some efficient C idioms have no MATLAB counterpart.

There are several approaches that compile a subset of MATLAB to other languages, namely Fortran (DeRose 1996) or C (Prasad et al. 2011). MATISSE (Bispo et al. 2013), a compiler framework on top of which we built a MATLAB-to-C compiler, is capable of generating efficient C code for a non-trivial subset of MATLAB, and we are continuously working to expand this subset. Other goals include generating readable C code, support for Aspect-Oriented Programming (AOP) concepts (Kiczales et al. 1997) and offering fine-grained control over the generated C code, such as control over whether dynamic memory allocation is used. MATISSE also features a working prototype for a MATLAB-to-OpenCL code generator (Bispo et al. 2015b).

As we expanded our supported subset of MATLAB and started working on more demanding benchmarks and features, we found ourselves increasingly constrained by the architecture of our compiler. In order to improve the compiler, we redesigned some portions of the C backend by introducing an SSA intermediate representation, on which most of our analyses and optimizations are now based. We also revamped various components to use this representation.

In this paper we introduce several transformations used in our system, and compare our automatically generated code with an equivalent handwritten version in C. This paper makes the following contributions:

- It describes our SSA approach for MATLAB-to-C compilation;
- An SSA-based iterative type inference algorithm;
- Important optimizations necessary to generate efficient low-level code, namely the use of a Z3-based solver to remove redundant runtime checks, and how to transform MATLAB expressions into efficient loops;
- A set of improvements to SSA-to-C code generation, to ensure that the generated code is efficient and to account for our MATLAB extensions;
- An evaluation of how automatically generated code compares to handwritten C code.

The remainder of this paper is structured as follows. Section 2 describes our SSA representation for MATLAB. Section 3 describes the most important SSA optimizations MATISSE applies

in order to generate efficient code, including a type and shape inference algorithm. Section 4 describes how to obtain C code from this IR and Section 5 evaluates our compiler based on a number of metrics. Section 6 describes related work and Section 7 concludes this paper and suggests future work.

2. Matrix-Based SSA

Previous versions of MATISSE (Bispo et al. 2015a) had two intermediate representations: an AST based on MATLAB (which we call "MATLAB IR") and another tree-based representation that attempts to closely mimic C concepts and semantics, solely used for C code generation (which we call the "C IR"). The MATLAB IR was directly converted to C IR statement by statement, and type inference occurred during this process. This limited the scope of many analysis and transformations, including type inference. This is undesirable because we often want to apply function-wide optimizations or code transformations that rely on type information but for which the C IR is too low-level.

To solve this problem, we developed a new SSA representation to MATISSE, between the MATLAB IR and the C IR, and built a pass-based system for it. The type inference mechanism, the optimizations and the C IR generator are now based on the SSA IR. Because the IR is built before type inference, it has both typed and untyped variants.

Figure 2 shows the SSA representation for the simple MATLAB program presented in Figure 1, before type inference has been applied. In this IR, each function is composed of one or more blocks, and each block has zero or more instructions. Each instruction performs a simple operation, such as a matrix set or a function call, so complex statements are often translated to multiple instructions that use temporary variables. The control flow mechanisms of this IR are close to their MATLAB counterparts, so concepts like for loops and breaks have explicit instructions, and unconditional jumps do not exist. In the SSA, matrices are first-class values, meaning that any modification (such as a matrix set) must be assigned to a new variable. Conceptually, every matrix set copies the entire matrix before setting the appropriate value, but in practice the variable allocator prevents this from happening by assigning multiple versions of the matrix to the same variable whenever possible. MATLAB operators (e.g., +, -, /) are converted to the equivalent function calls.

Variables that start with a $ symbol are temporaries. The remaining variables have a direct equivalent in the original MATLAB source code, with the variable prefix before the $ being the corresponding MATLAB name (e.g., `xyz$1` corresponds to `xyz`). Return values are stored in variables ending in `$ret`. The SSA IR has instructions to efficiently express both high-level MATLAB idioms (such as matrix sets that may resize matrices, `matrix_set`) and lower-level optimized constructs (e.g., `simple_set`, that represents matrix sets that are known to be in range, and therefore never resize the matrix and do not need additional checks).

The compiler has also been improved in several ways. It now includes an iterative type and shape inference system, automatic performance diagnostics that statically indicate parts of the program that may be inefficient, and support for matrix resizes through out-of-range sets (e.g., `A(4) = 1;` where A has fewer than 4 elements or has not been previously declared at all).

3. Optimizations

We describe some of our most important optimizations introduced with the SSA system.

3.1 Improvements to Type and Shape Inference

Type and shape inference is essential because MATISSE requires accurate type information in order to generate efficient C. Many

```
function y = f()
   y = zeros(1, 10);
   for i = 1:10,
      y(1, i) = i + 1;
   end
end
```

Figure 1. Simple MATLAB program.

```
block #0:
   $zeros_arg1$1 = 1
   $zeros_arg2$1 = 10
   y$2 = untyped_call zeros $zeros_arg1$1,
$zeros_arg2$1
   $start$1 = 1
   $interval$1 = 1
   $end$1 = 10
   for $start$1, $interval$1, $end$1, #1, #2
block #1:
   y$3 = phi #0:y$2, #1:y$4
   i$2 = iter
   $plus_arg2$1 = 1
   $plus$1 = untyped_call plus i$2,
$plus_arg2$1
   $y_index1$1 = 1
   y$4 = matrix_set y$3, $y_index1$1, i$2,
$plus$1
block #2:
   y$ret = phi #0:y$2, #1:y$4
```

Figure 2. SSA representation of the program shown in Figure 1.

MATLAB operations have different semantics depending on the types of the inputs, so knowing which ones should be used helps reducing the number of runtime checks. With the transition to the SSA system, we made substantial changes to the type inference system.

First of all, it is now possible for a MATLAB variable to change types in a function, provided it has only a single type at each point of the program. This is possible because each MATLAB variable can have multiple corresponding SSA variables, and each of those can have its own type.

For code without loops, the new type inference system behaves similarly to the old one. Each instruction type has an associated type inference rule to determine the types of the output variables. When a ϕ instruction for a branch is found, then the types of the input variables are combined and the result is used for the output variable. When equal variables are combined, the result is an identical type. If compatible variable types are combined (e.g., a 2D double matrix and a 3D double matrix), the result is the general variable type that covers all combined types (e.g., a double matrix with an unknown number of dimensions). If the types are incompatible, then the type inference emits an error.

When the type inference pass reaches a loop, it visits its contents multiple times until the result of the inference stabilizes. To do so, it uses a queue *pending start blocks*. For every start block N, there is a list of inferred variable types (the *type context for* N) and a list where it stores the results of the algorithm (the *end data*). Initially, the *pending start blocks* queue contains only the ID of the block

that contains the loop. The instructions in the loop are then visited iteratively until the *pending start blocks* queue is empty:

- Remove an element from the *pending start blocks* queue. We refer to this as the loop *entry point*.
- The current variable types are those in the context for the *entry point*.
- When a ϕ instruction is found at the beginning of the loop, set the type of the output variable to be the same as the variable coming from the loop *entry point*.
- Every time a `break` instruction is found, add the current variable type information to the *end data*.
- Every time a `continue` instruction is found, add the current variable information to the *type context* for the block containing the `continue` instruction. If this results in any changes to that context, add the ID of the block containing the `continue` instruction to the *pending start blocks*. If the current loop is a `for` loop (as opposed to a `while`), then also perform the operation described for `break` instructions.
- Treat the end of the loop as an implicit `continue` instruction.

When adding type information to a *type context* that does not exist, the compiler creates a new *type context* with that information. When a variable type is added to a *type context*, the compiler combines the new type with the type that was previously in that context and use the result as the new type for that context.

3.2 Loop Conversion Passes

The previous version of MATISSE implemented an optimization to convert element-wise operations, such as matrix addition, into for loops. Our SSA system also performs this transformation, but we broke it into a larger set of interacting transformations, rather than a single one. We call this group of transformations "Loop Conversion Passes":

1. Conversion of certain instructions, such as range matrix accesses (e.g., `A(:, 1:4)`), element-wise (e.g., `A + B`) and reduction (e.g., `sum(X)` function calls) to loops;
2. Loop fusion;
3. Elimination of reads after writes to the same address;
4. ~~Dead Code Elimination;~~
5. Loop matrix dependency elimination.

Step 1 translates expressions such as `X = A(:);`, `Y = A + B;` or `x = sum(A)` (where A is a 1D matrix) into the equivalent for loops. At the SSA stage the concept of statements no longer exists, so this captures operations that appear in subexpressions such as `f(A + B)`. However, complex expressions will result in more loops that necessary. For instance, the statement `x = sum(A(:) + B .* C)` will generate 4 loops: one for the range access, one for the addition, one for the multiplication and one for the sum. Additionally, we will also allocate three unnecessary temporary matrices (one for `A(:)`, one for `A(:) + B` and one for `A(:) + B .* C`).

Step 2 merges multiple for loops. For loops can be fused if a number of conditions are met, notably related to side-effects, number of iterations and existence of instructions such as `break`. Our loop fusion pass is capable of merging loops with different depths (e.g., a 2D loop with a 1D loop).

Figure 3 shows the equivalent MATLAB code for the statement `x = A(:) + B .* C`, after loop fusion. We can see that now there is only one loop, and we still have temporary matrices which are written to, and then immediately read at the same index. Step 3 replaces the reads to the temporary matrices with an access to the original value, so that Step 4 can delete the temporary matrices.

```
for i = 1:iters,
        tmp1(i) = A(i);
        tmp2(i) = tmp1(i) + B(i);
        x(i) = tmp2(i) .* C(i);
end
```

Figure 3. MATLAB code representing the behavior of the IR code for `x = A(:) + B .* C` after loop fusion.

At this point, the SSA code for statements such as `A(i, :) = A(i - 1, :);` consists of a single for loop. However, the matrix read still references the version of the matrix from before the loop, as in `A_in_loop(i, j) = A_before_loop(i - 1, j)`. In this case, the variable allocator can't assign both variables to the same C matrix because their lifetimes intersect (see Section 4). This results in unnecessary matrix copies and suboptimal performance. To avoid this, we apply Step 5 (Loop Matrix Dependency Elimination) that replaces `A_before_loop` with `A_in_loop` whenever it detects that doing so is valid.

3.3 Use of the Z3 Theorem Prover

As we expanded the supported MATLAB subset, a number of runtime checks are necessary to ensure correctness. For instance, every matrix set can potentially trigger a matrix resize so we must check whether an index is out-of-range. Even if those resizes never happen at runtime, the mere presence of the checks in the code can take a significant amount of time and inhibit valuable compiler optimizations.

We handle these situations in two manners: MATISSE features an "Unchecked" mode that turns off most of these checks. This mode can be applied per-function, or to an entire program at once. Additionally, we attempt to statically check whether certain checks are necessary at all using a solver. We try to remove as many runtime checks as possible, and convert complex SSA instructions (e.g., `matrix_set`) into simpler ones (e.g., `simple_set`).

Our solver code is separated in two parts: The Shape Solver and the Scalar Solver. The shape solver keeps track of the relative shapes of variables and finds statements such as `A = zeros(N, M);` to discover facts such as `size(A, 1) == N`. It also identifies when multiple matrices have identical sizes and that certain variables contain the size of other matrices (e.g., `x = size(A);`). However, the shape solver is unable to know how scalar values relate to each other. For those tasks, it uses the scalar solver.

The scalar solver receives function-wide information such as `i = a - 2` to determine whether statements such as `i < a` are necessarily true. MATISSE features two scalar solvers: a fast naïve solver developed by us for test purposes, and a second higher quality solver based on third-party libraries. We initially considered using Symja (Symja 2016), but we found it unsuitable for our purposes. For instance, it failed to identify that `a < a + 1` or that `a < b && b < c` implies `a < c`.

For these reasons, we decided to try Z3 (De Moura and Bjørner 2008), a full-fledged theorem prover by Microsoft Research that can be used to check theorems for satisfiability. In MATISSE, we want to determine whether certain expressions are *necessary*, not *satisfiable* (possible). To test an expression X, we assert $\neg X$ and see if the result is satisfiable. If it is not, we know that X is necessarily true.

We only use a small subset of Z3's features. Notably, we do not currently use any of Z3's array or bitvector operations. In addition, we rely on Z3's soft timeouts to prevent excessive compilation times.

3.4 Matrix Preallocation

MATLAB does not require programmers to initialize matrices before writing values to them, but doing so can still be tremendously beneficial for performance reasons. If a matrix set refers to a position out of index, then a resize operation is triggered. If this happens inside a loop, this can have a substantial negative impact on performance. The MathWorks Code Analyzer identifies and warns against this performance anti-pattern (Shure 2012) but, nevertheless, it still appears in some examples, such as the *Disparity* benchmark mentioned in Section 5.

MATISSE automatically identifies some simple variants of this anti-pattern and preallocates the matrix, with significant performance improvements.

3.5 Pass by Reference

When MATLAB matrices are used as function arguments, they are passed by value. Conceptually, this means that if the matrix is modified by the function, such changes are not visible to the caller, which still has access to the original matrix. We handle this case by having the callee copy any matrix arguments that may be modified in its body.

This approach prevents some copies and is compatible with MATLAB semantics, but it's not sufficient to match C performance. We extended MATISSE to identify directive comments of the form `%!by_ref VarName`, to identify which arguments should be passed by reference. We wanted this approach to be both compatible with MATLAB, consistent with our SSA IR semantics and resilient to user errors. Based on these constraints, we designed the directive to work as follows:

- If a function input `X` is passed by reference, then there must be a single function output with the same name (the order of the inputs/outputs does not matter).
- If a function has a `%!by_ref` output, then whenever it is called, its `nargouts` must be large enough to cover that output.
- The inferred type of the output must be compatible (in terms of C signature) with the type of the corresponding input.
- Arguments passed by reference in MATLAB *must* be simple variables, not complex expressions.
- In the caller, the inputs passed by reference and their matching outputs must correspond to the same MATLAB variable.
- Currently, only matrix types are supported. It does not make sense to pass scalars by reference as there is no gain to be had.

Even then, it is not always possible to avoid using a separate matrix. We emit a performance warning when the variable allocator is unable to properly handle `%!by_ref`, but we ensure that the generated code is still correct in these cases.

The modified code still works correctly in MATLAB, but users can use this feature to reduce the number of matrix allocations in the C code that MATISSE generates.

3.6 Other Optimizations

MATISSE includes a number of other optimizations, including:

- Elimination of branches, when the boolean value of the condition is known at compile time.
- Elimination of loops with 0 or 1 iterations. We effectively perform a full loop unroll in these cases.
- Loop Interchange for some simple cases where loop iterations are only used for matrix accesses, and each iteration does not depend on the results of any other iterations.
- Bounds check motion for `matrix_get` instructions inside loops.
- Reduction operations [!!! Adicionar description, "feature" que est na tabela 1]

In total, the MATISSE MATLAB-to-C compiler now has more than 30 different optimization passes, in addition to various other support analyses and transformations.

3.7 Comparison of MATISSE versions

Table 1 shows how the current version of MATISSE compares to previous ones. Most optimizations are still supported, the exceptions being weak types and matrix views (Bispo et al. 2015a). Weak types is a technique that allows some limited multi-pass type inference behavior when performing only a single-pass. Since MATISSE now uses iterative type-inference, this technique is no longer needed. Matrix views are less useful now due to the improvements described in Section 3.2, but they could still yield improvements in some examples, so we may reintroduce this optimization in the future.

Overall, we now support a larger subset of MATLAB and feature a wider range of optimizations that cover most of the cases that the previous system was able to handle, in addition to several new ones.

4. Code Generation

After applying the SSA passes, we generate C code. We first convert some of our more complicated SSA instructions into their simpler counterparts. Then, we determine which C variables correspond to which SSA variables and eliminate ϕ nodes. To do so, we use an algorithm based on (Boissinot et al. 2009), which we adapted to better fit our needs. Since this stage is not our bottleneck, we skipped the performance improvements aimed at JIT compilers and implemented a simpler but slower version of the algorithm.

1. We convert our IR to CSSA form by adding parallel copy instructions, as described in Method I of (Sreedhar et al. 1999);
2. We then determine which SSA variables should be grouped into a single C variable, using a method which we describe in more detail later in this section;
3. We generate a C variable name for each group of SSA variables;
4. We generate the C IR one instruction at a time;
5. Once the C IR for all functions is constructed, we apply a set of code cleanup passes;
6. Finally, we generate the C code from the C IR.

In order to determine which SSA variables can be grouped together, we construct the liveness sets for each SSA instruction and build the interference graph. This is similar to the approach described by Boissinot et al. (Boissinot et al. 2009), but with an important exception: output variables of function calls may sometimes interfere with the inputs, even if the input variables are never used again in the caller function.

To understand why this happens, consider a C function with signature `void f(int x[10], int y[10])`, where `x` is the input of the function and `y` is the output. Without examining the implementation of `f`, we cannot know if it safe to assign these two variables to the same C one. As such, we assume that input variables are live at the end of a function call unless they are scalars or marked as `%!by_ref`.

Once we know the interference graph, we create groups of SSA variables that should be assigned together:

Feature	ARRAY'14 (Bispo et al. 2014)	ARRAY'15 (Bispo et al. 2015a)	SSA System
Intermediate Representations (IRs)	MATLAB IR C IR	MATLAB IR C IR	MATLAB IR SSA-based IR C IR
Type and Shape Inference	During MATLAB IR to C IR translation Forward-only		Separate phase on SSA IR Iterative (Forward with backtracking)
Transformation of Element-wise Operations	No	Statement-based	Yes, including sub-expressions
Support for Weak Types	No	Yes	No
Support for Matrix Views	No	Yes	No
Transformation of Reduction Operations	No	No	Yes
Loop Fusion	No	No	Yes
Loop Interchange	No	No	Simple cases only
Algebraic Analysis	No	Based on Symja	Based on Z3
BLAS support	No	Yes	Yes
Support for matrix resizing	No	No	Optional
Matrix preallocator	N/A	N/A	Trivial loops only

Table 1. Comparison of MATISSE versions

1. First, we assign all variables in ϕ nodes to a single group, as required by Boissinot et al.'s algorithm (Boissinot et al. 2009).
2. Then, we visit all instructions, and try to group variables in assignments, parallel copies and `simple_set` instructions.
3. Finally, we visit all instructions again, and try to group variables in `matrix_set` instructions and `%!by_ref` function arguments.

Ensuring that matrix variables are grouped together is very important, as matrix copies are expensive operations. We are not concerned about grouping scalar variables, as C compilers will typically perform their own allocations.

Once we have groups of SSA variables, we assign a C variable name to each group. We try to use the original MATLAB names when possible. For temporary variables, we use a name based on its role (e.g., `iter$1` has role `iter`). If two C variables have the same name, or if a variable has an invalid name (i.e. a C keyword or library function), we add a numeric suffix to avoid any conflicts.

At this stage the C code has a number of readability issues, so we apply a few cleanup passes over the C IR. The construction of C code from C IR is mostly unmodified from previous MATISSE versions, with only minor adjustments to account for the existence of `%!by_ref` parameters.

5. Experimental Results

We present two sets of results, 1) a set of benchmarks that were used in a previous version of MATISSE (Bispo et al. 2015a) and 2) an in-depth analysis using the *Disparity* benchmark of the San Diego Vision Benchmark Suite (SDVBS) (Venkata et al. 2009). The SDVBS includes both MATLAB and C versions of the same programs. The C version is not a direct translation of the MATLAB code, giving us the ability to compare automatically generated code with handwritten versions.

All MATLAB and C versions were tested on a desktop computer running Windows 10 Enterprise 64-bits, with an AMD A10-7850K CPU and 8GB of DDR3 RAM. We tested the program with the Full HD image included in the *Disparity* benchmark.

Our raw results and code versions for *Disparity* can be obtained at `http://specs.fe.up.pt/publications/array16.zip`.

5.1 Benchmarks in previous versions of MATISSE

~~For~~ our previous benchmark suite, ~~we got mixed results. Although~~ some benchmarks are now faster, other exhibited performance regressions or failed to compile.

For the benchmarks we were able to test, the new system was faster in 9 ~~examples~~, but slower in 11. In particular, in most of the cases where the new system outperformed the old one, the difference was small, whereas in most cases where the old system outperformed the SSA one, ~~the difference was substantial~~.

This is unsurprising because these benchmarks have received substantial attention when optimizing our old system, and we have not yet done this for the SSA system.

5.2 Analysis of the MATLAB version of Disparity

The MATLAB version of the *Disparity* benchmark is composed of 3 files: `script_run_profile.m`, a function that behaves as an entry point for the program, `getDisparity.m`, that contains the actual algorithm implementation, and `refineDisparity.m`. This last file does not appear to be used by `script_run_profile.m` and as such was excluded from further analysis. We modified `script_run_profile.m` to be better integrated with our testing environment, but no other code was modified, except in the code versions explicitly labelled as such.

The original `getDisparity.m` file has a total of 47 non-empty lines of code, of which 3 are comments. This file relies on only a few built-in MATLAB functions. Aside from matrix allocation, type casting and built-in operators, only `padarray`, `min` and `size` are used.

According to the MATLAB R2015a profiler, the MATLAB version takes approximately 261.8s to complete. Nearly all of that time (91.5%) is spent on a single line of code, a matrix set in a 2D loop (corresponding to the C function `computeSAD`). This happens because the matrix was not allocated before the loop, so the matrix will be continuously resized, as explained in Subsection 3.4.

We then added a single line of code to allocate the entire matrix before the loop. This version (`matlab_modified`) takes approximately 28.0s to run. We tried some other approaches to reduce the execution time in MATLAB and found that the best approach (`matlab_modified3`) was to remove the `computeSAD` loop altogether and use matrix operations instead. This last version takes about 12.2s to run.

5.3 C versions

The San Diego Vision Benchmark Suite includes handwritten C versions for each benchmark. We refer to this the "Handwritten" version. Whereas the MATLAB version of *disparity* uses double-precision floats, the Handwritten C version uses integers and single-precision floats instead. To measure the impact of this difference,

we modified the Handwritten version to use double-precision floats as well. We call this modified version the "Handwritten (Double)" version.

Based on the techniques described previously in this paper, we have obtained 6 versions derived from MATLAB:

- Original MATLAB (w/o Z3) - C version generated from the original MATLAB code, with runtime checks enabled and using a simple placeholder solver[1].
- Original MATLAB (w/ Z3) - Similar to "w/o Z3", but using a solver based on Z3.
- Original MATLAB (Unchecked) - C version generated from the original MATLAB code, with fewer runtime checks and using a solver based on Z3.
- Modified 1 - Similar to "Unchecked", but the MATLAB code was modified so that the generated code was more efficient, as described later in this section.
- Modified 2 - Similar to "Modified 1", but using the `%!by_ref` directive described earlier to reduce the number of memory allocations. This version is still mostly compatible with MATLAB, though.
- Manually Improved A - Based on "Modified 2", but the generated C code was modified to remove 4 unnecessary matrix allocations.
- Manually Improved B - Based on "Manually Improved B", but with a manual application of a loop interchange.

The performance improvements to the MATLAB version described in the previous section were not used here. As such, the "Original" MATLAB version refers to the code without the explicit matrix allocation.

Version "Modified 1" has the following improvements over "Unchecked": 1) replacement of a 2D loop with an equivalent 1D, 2) MATLAB code modifications to more closely resemble the C code, 3) removed calculations for outputs that are known to be unused, and 4) computation of partial results on each iteration of the program, instead of keeping track of the output of every iteration and performing a full computation near the end of the program. This final change did introduce a MATISSE-exclusive feature (matrix allocation functions without default values), but creating a fallback for MATLAB is trivial, and MATISSE provides a MATLAB implementation of the function. This is also the only part of the "Modified 2" version that is not compatible with MATLAB.

Table 2 shows how many lines of MATLAB code were changed in the `getDisparity.m` file in order to obtain the 2 modified versions. All line counts exclude comments and empty lines, but include directives. The total number of changed lines is substantial (more than half of the total lines of code), but a most of these (23) can be attributed to a single change: the replacement of the *min* function call with the C-like partial computation approach. Regardless, all 3 versions have the same number of functions in this file (and no other files were changed, added or removed).

The Manually Improved versions are based on the "Modified 2" C code, but with manual changes to evaluate optimizations that we could add to MATISSE in the future, to evaluate whether those optimizations are worth the effort.

5.4 Analysis of the C code

We analyzed and extracted metrics from the original handwritten version, as well as the MATISSE-generated C files. The results can be seen in Table 3. A function is considered to be part of the computation ("Comp.") if it is part of the specific operations required to compute the result. Functions that serve a generic system purpose (such as memory allocation), that exist in MATLAB (such as `padarray`) or that are part of the MATISSE system (such as `MATISSE_raw_ind2sub`) are considered to be "Support".

Functions to import data files or to perform benchmarking were ignored, and so were calls to external C functions, such as standard library or Windows API calls. Only reachable functions were counted, i.e., functions that are never called directly or indirectly by the main function were excluded from this table. We also excluded the `main` function itself, as it contains substantial data loading and benchmarking code that we did not consider to be relevant for this analysis.

The handwritten versions have substantially fewer "support" functions than the generated ones. Part of the reason is that MATLAB functions such as `size` are implemented as C functions (one per used matrix type, with more combinations for each of the variants that this MATLAB function has), whereas this does not happen in the handwritten versions. Additionally, matrix allocation in MATISSE causes the generation of several C functions, where the handwritten version uses only 6.

The handwritten version also has fewer lines of code for the computation part of the program than the generated versions. The two main causes of this are:

- MATISSE-generated variable declarations consist of a single variable per line. In contrast, the handwritten version declares multiple variables of the same type in a single line.
- Several subexpressions are assigned to temporary variables in MATISSE, whereas in the handwritten version they tend to be part of more complex expressions.

Metric	Original	Modified 1	Modified 2
Total Code Lines	44	52	57
Added, Removed or Modified Lines	0	32	45
Directives	0	0	2
Total Functions (getDisparity.m)	3	3	3

Table 2. Differences between MATLAB versions.

Version	Functions		Lines of Code
	Comp.	Support	(*.c Comp. Files)
Handwritten	6	8	166
Handwritten (Double)	6	5	166
Original MATLAB (w/o Z3)	5	43	826
Original MATLAB (w/ Z3)	5	42	514
Original MATLAB (Unchecked)	5	42	481
Modified 1	4	33	300
Modified 2	4	28	301
Manually Improved A	4	21	258
Manually Improved B	4	21	258

Table 3. Comparison of the various C versions, in terms of number of functions and non-empty lines of code.

[1] An online version of MATISSE that generates the equivalent of this version can be tested at `http://specs.fe.up.pt/tools/matisse-new/`.

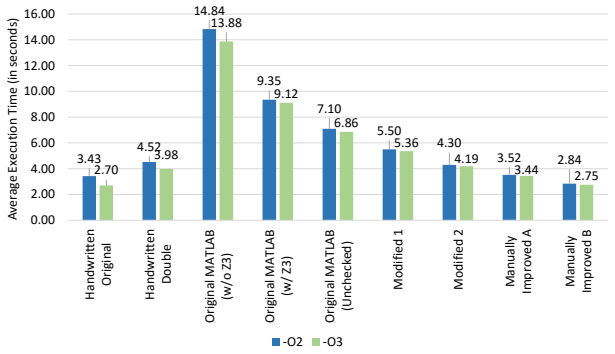

Figure 4. Average execution times for the various versions, running on a desktop computer.

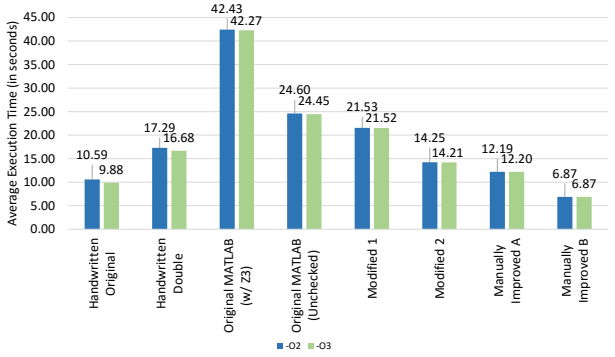

Figure 5. Average execution times for the various versions, running on an Odroid device.

5.5 Performance Analysis

We compiled the code with MinGW-W64, a 64-bits version of MinGW based on GCC 4.9.2, with `-O2` and `-O3`. We ran each version 30 times and computed the average and the standard deviation of the execution times, as measured by the Windows `QueryPerformanceCounter` function. We found the estimated error for 95% confidence to be negligible.

The average execution times for the desktop system can be seen in Figure 4. As we can see, we can achieve execution times around 1.54 times slower than the handwritten version without changes to the MATLAB code (Unchecked, with `-O3`). If we are willing to optimize the MATLAB code, even to the point of using MATISSE-exclusive features, then MATISSE can generate code that is about 55% slower than the handwritten version (Modified 2, with `-O3`). There is still potential for further MATISSE improvements, as the optimizations proposed in the manually improved versions should be possible to implement in the MATISSE compiler, which would further lower the slowdown to approximately 1%, even though the code generated by MATISSE is using doubles where the handwritten version uses floats and integers. When we modify the handwritten version to also use doubles, we see that MATISSE is already very close (Modified 2 compared to Handwritten Double, with any optimization level) and with some tweaks, the generated code is able to outperform the handwritten version.

Additionally, we tested the C versions of *Disparity* on an Odroid XU+E, containing an Exynos5 Octa CPU (with 4 Cortex A15 and 4 Cortex A7 cores) and 2 GB LPDDR3 RAM, running Ubuntu 14.04.2 LTS with GCC 4.8.2. The results can be seen in Figure 5.

	SAD		integralImage2D		
	Compute	Final	#1	#2	#3
Handwritten (Original)	Yes	No	Yes	Yes	No
Handwritten (Double)	Yes	No	Yes	Yes	No
Modified 1	Yes	Yes	Yes	No	Yes
Modified 2	Yes	Yes	N/A	No	Yes
Manually Improved A	Yes	Yes	N/A	No	Yes
Manually Improved B	Yes	Yes	N/A	No	Yes

Table 4. Vectorization of loops in various code versions.

The Original MATLAB version without Z3 could not be executed due to lack of memory on the device, as this version generated too many temporary variables. On this device we are able to generate code (Modified 2) that is faster than the Handwritten (Double) version. We can also see that on this device, the impact of `-O3` is negligible, and loop interchange has a very substantial impact.

We have also checked whether the C compiler was able to vectorize the loops of our best 4 generated versions, as well as the handwritten one, using GCC's `-fopt-info-vec` diagnostics. The results can be seen in Table 4. Even though there is not a 1-to-1 function correspondence between the handwritten version and the generated versions, we can still identify which handwritten loops match which generated loops. We focused on three C functions - computeSAD, finalSAD and integralImage2D2D - which correspond roughly to the correlateSAD and integralImage2D MATLAB functions. Additionally, we ignored loops present in the generated version but not in the handwritten one. In `integralImage2D`, the `#1`, `#2` and `#3` refer to the 3 loops in the C version. For multi-dimensional loops, the vectorization results refer to the inner-most loops. We include the results for `-O3` only, as GCC does not vectorize by default on `-O2`.

GCC is able to vectorize the MATISSE generated equivalent to finalSAD, even though it could not vectorize the handwritten version. The loops seem very similar, with the most significant difference being that the generated versions use double instead of single-precision floats and that some values are computed outside the innermost loop (though loop invariant code motion should be able to do the same), so it is not clear what is causing this.

The first loop in `integralImage2D` has no equivalent in the last 3 versions as it was made redundant due to changes introduced by `%!by_ref`. The reason the loops in integralImage2D that are vectorized in the generated versions are different from the handwritten versions is because MATLAB is column-major, whereas C is row-major. Since each of the loops reads values along a dimension, the two loops effectively swap places in the generated versions. The decision to use double-precision in the handwritten version has no impact on vectorization.

Overall, we consider these results to be positive, since we were able to obtain results very close to the handwritten version, using automatically generated code and a few modifications in MATLAB. If we are willing to manually tweak the generated C code, it is possible to go further than the handwritten version (1.4-1.6 times speedup).

6. Related Work

The compilation of MATLAB to lower-level languages has been the subject of multiple research projects. One of these projects is FALCON (DeRose 1996), which compiles MATLAB to FOR-

TRAN, using an SSA IR. The way they deal with matrix sets is similar to our own, but their IR has a number of substantial differences, notably related to its structure (AST-based as opposed to block/instruction-based) and the way ϕ nodes are handled.

MATLAB Coder (MathWorks 2016b) is a MATLAB-to-C compiler by MathWorks that is capable of generating efficient C code from a large subset of MATLAB. However, there is little to no public information about which optimizations they apply.

MEGHA (Prasad et al. 2011) is a compiler capable of generating C++ or CUDA source code from MATLAB programs. They also use an SSA-based IR but with some substantial differences. They do not create new variables for matrix sets (which suggests that their SSA works only for scalars) and they optimize the insertion of ϕ nodes to ensure that these only happen for live variables. In contrast, we found that creating new variables on matrix sets is not a problem, provided that the final variable allocation code is sufficiently aggressive. We do not take particular care about ϕ node insertion, as any redundant ϕ nodes are removed by ~~dead code elimination~~.

Sci2C (David et al. 2016) is a Scilab to C compiler. Sci2C works by parsing directive comments that indicate the type, size and precision of variables. Aside from the different source language, Sci2C differs from MATISSE in that it does not support dynamically sized matrices. Additionally, in MATISSE all directives are strictly optional.

7. Conclusion

In this paper, we described improvements to our MATLAB-to-C compiler framework, including an SSA-based IR, an iterative type inference algorithm, a set of optimizations for improving performance and our code generation mechanism.

As part of our evaluation, we used a representative benchmark from the San Diego Vision Benchmark Suite, and we compared the results of MATISSE to handwritten C code in a number of metrics, notably performance and code size, and examined which optimizations we should implement in the future to reach the level of performance of the handwritten code.

We found that automatically generated C code can match or even outperform handwritten code, provided that a number of important optimizations are implemented and certain modifications are made to the MATLAB code.

In the future, we intend to improve our optimizers to reach the level of performance of the handwritten versions, so that the generated code can be comparable to the manually improved code versions. Additionally, we intend to expand our analysis to more applications of the San Diego's Vision Benchmark Suite, to see which additional optimizations are needed to achieve good performance in other examples. We will also analyze our previous benchmarks to identify and fix the causes of performance regressions. Finally, we will apply the lessons we learned to our OpenCL backend, so that MATISSE can generate parallel code compatible with GPUs and FPGAs.

Acknowledgments

This research has been supported by the Portuguese *Fundação para a Ciência e Tecnologia* (FCT), through a PhD scholarship (PD/BD/105804/2014).

References

J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. P. Cardoso, and P. C. Diniz. The MATISSE MATLAB Compiler - A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms. In *IEEE International Conference on Industrial Informatics (INDIN2013)*, Bochum, Germany, 29-31 July 2013.

J. Bispo, L. Reis, and J. M. P. Cardoso. Multi-Target C Code Generation from MATLAB. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 95:95–95:100, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8.

J. Bispo, L. Reis, and J. M. P. Cardoso. Techniques for Efficient MATLAB-to-C Compilation. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 7–12, New York, NY, USA, 2015a. ACM.

J. Bispo, L. Reis, and J. M. P. Cardoso. C and OpenCL Generation from MATLAB. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1315–1320, New York, NY, USA, 2015b. ACM.

B. Boissinot, A. Darte, F. Rastello, B. D. de Dinechin, and C. Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

C. David, A. Cornet, and M. Baudin. Scilab 2 C - Translate Scilab code into C code. `http://forge.scilab.org/index.php/p/scilab2c/`, 2016. Accessed: March 25th, 2016.

L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

L. A. DeRose. Compiler Techniques for MATLAB Programs. Technical report, Champaign, IL, USA, 1996.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

MathWorks. MATLAB - The Language of Technical COmputing. `http://www.mathworks.com/products/matlab/`, 2016a. Accessed: March 25th, 2016.

MathWorks. MATLAB Coder - Generate C and C++ code from MATLAB code. `http://www.mathworks.com/products/matlab-coder/`, 2016b. Accessed: March 29th, 2016.

A. Prasad, J. Anantpur, and R. Govindarajan. Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. *SIGPLAN Not.*, 46(6):152–163, June 2011. ISSN 0362-1340.

L. Shure. Understanding Array Preallocation - Loren on the Art of MATLAB. `http://blogs.mathworks.com/loren/2012/11/29/understanding-array-preallocation/`, 2012. Accessed: March 23rd, 2016.

V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form. In A. Cortesi and G. Filé, editors, *Static Analysis: 6th International Symposium, SAS'99 Venice, Italy, September 22–24, 1999 Proceedings*, pages 194–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

Symja. Symja - Java Computer Algebra Library. `https://bitbucket.org/axelclk/symja_android_library/wiki/Home`, 2016. Accessed: March 23rd, 2016.

S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society.